Naval Research Laboratory
Washington, DC 20375-5000

NRL Report 9187

31073

Security Architecture for a Secure Military Message System

MARK R. CORNWELL AND ANDREW P. MOORE

Center for Secure Information Technology
Information Technology Division

April 28, 1989

Approved for public release; distribution unlimited.

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY 2b. DECLASSIFICATION / DOWNGRADING SCHEDULE	3. DISTRIBUTION / AVAILABILITY OF REPORT Approved for public release; distribution unlimited.
4. PERFORMING ORGANIZATION REPORT NUMBER(S) NRL Report 9187	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Naval Research Laboratory 6b. OFFICE SYMBOL *(If applicable)* Code 5540	7a. NAME OF MONITORING ORGANIZATION
6c. ADDRESS *(City, State, and ZIP Code)* Washington, DC 20375-5000	7b. ADDRESS *(City, State, and ZIP Code)*
8a. NAME OF FUNDING / SPONSORING ORGANIZATION Space and Naval Warfare Systems Command 8b. OFFICE SYMBOL *(If applicable)*	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS *(City, State, and ZIP Code)* Washington, DC 20363-5100	10. SOURCE OF FUNDING NUMBERS

10. SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT ACCESSION NO.
35167G	68003		DN 880-204

11. TITLE *(Include Security Classification)*
Security Architecture for a Secure Military Message System

12. PERSONAL AUTHOR(S)
Cornwell, M. R. and Moore, A. P.

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT *(Year, Month, Day)*	15. PAGE COUNT
Interim	FROM 1/88 TO 8/88	1989 April 28	39

16. SUPPLEMENTARY NOTATION

17. COSATI CODES

FIELD	GROUP	SUB-GROUP

18. SUBJECT TERMS *(Continue on reverse if necessary and identify by block number)*
Computer security
Message system
Software engineering
Application-based security model

19. ABSTRACT *(Continue on reverse if necessary and identify by block number)*

This document defines a security architecture used in the Secure Military Message System (SMMS) project at the Naval Research Laboratory. The goal of this architecture is to provide high assurance that the full-scale prototype message system being constructed as a part of this project is secure. It presents design decisions and shows why a system built according to these decisions will conform to a formal model of security for processing military messages.

20. DISTRIBUTION / AVAILABILITY OF ABSTRACT
☒ UNCLASSIFIED/UNLIMITED □ SAME AS RPT. □ DTIC USERS

21. ABSTRACT SECURITY CLASSIFICATION
UNCLASSIFIED

22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE *(Include Area Code)*	22c. OFFICE SYMBOL
Andrew P. Moore	(202) 767-6698	Code 5543

Previous editions are obsolete. SECURITY CLASSIFICATION OF THIS PAGE

CONTENTS

SECURITY ARCHITECTURE FOR A SECURE MILITARY MESSAGE SYSTEM

INTRODUCTION

A *security architecture* describes the part of the system structure that provides security. It covers enough of the system structure to argue convincingly that the system is secure, details enough that the feasibility of a trustworthy implementation is evident, and tells no more about the system structure than is necessary for this purpose.

This report describes the security architecture used in the Secure Military Message System (SMMS) project at the Naval Research Laboratory. The goal of this architecture is to provide high assurance that the full-scale prototype message system being constructed as a part of this project is secure. It presents a set of design decisions and shows why a system built according to these decisions will conform to a formal model of security for the military message processing application. This report provides an easy review of the security design by collecting information in one place and leaving out information that does not bear on security. The SMMS designers use it to record decisions about the security design of the system and the security arguments that justify these decisions. Certifiers can use it to review the system structure and to evaluate the security arguments provided. To others, it serves as a useful example documenting a disciplined design methodology.

We outline the working security architecture being used in the design of a prototype military message system. It is certain that the architecture will continue to evolve as the prototyping process continues. The design is the result of several years of evolution and is our best effort to date to define an architecture that gives strong assurance of the security properties defined by the SMMS security model.

SECURITY MODEL

Application-Based Approach

The SMMS model is the product of an application-based approach to computer security that differs from conventional approaches based on the generic security model originally developed by Bell and LaPadula [1]. These approaches do not work well when applied to applications that look different from the ones the model implicitly assumes. The approach taken in deriving the SMMS model is intended to overcome the problems experienced in applying conventional generic models to such software applications, particularly to message systems.

Problems with Conventional Approaches

Experience indicates that there is a poor fit between conventional security models and the actual security requirements of many applications. This poor fit is exhibited in three ways. First, systems using conventional models often need to introduce mechanisms to circumvent security checking when

Manuscript approved August 29,1988.

it proves overly restrictive. Second, conventional models often are not understood by the users, causing the users to misuse the system in ways that compromise security. Third, often security requirements specific to application are not included in the conventional models.

Mechanisms for Circumventing Security Policy

When building applications using conventional models, it is found that reasonable and necessary operations cannot be performed without violating the model. To perform these operations, one traditionally introduces *trusted processes*, which are exempt from normal security checking. Since the trusted processes are not constrained by the security model as are the untrusted processes, trusted processes have the potential to generate behavior that violates the security model. The use of trusted processes requires inspection of the implementing programs to ensure that they do not compromise some additional intuitive notion of security.

For example, initial versions of the Sigma message system [2] do not allow a user to create an unclassified reply to a classified message unless the user logs off and logs on again. Users want a single command that will extract information from particular fields of a classified message and insert them into a newly generated unclassified message. As long as the information from those fields is not classified, the operation is intuitively secure. To get around these restrictions, a trusted process was introduced to implement the reply command. As designers and maintainers confronted the realities of satisfying the users' demands for functionality, additional trusted processes were introduced.

The problem of complementing a security model with trusted processes is that the original model is no longer a valid description of the security policy the system is actually implementing. The actual policy is less restrictive than the model without trusted processes. The model, including trusted processes, has the potential for designers to specify trusted processes with no security constraints at all. Because of this, it is not restrictive enough. Some different, inexplicit policy stemming from an intuitive understanding of secure behavior is actually implemented. A better approach is to capture this intuitive understanding explicitly when the system requirements are defined and design the system for high assurance of the properties dictated by this understanding.

Clarity of Conventional Security Model

A problem arises when the end users do not understand the behavior implied by a conventional security model. Security models developed by computer scientists to describe secure operating systems are confusing to users who do not share the computer scientist's familiarity with operating system concepts. These users should not be required to have a conceptual model of how their application commands correspond to operating system processes and files to predict how the system will behave.

The Sigma message system also illustrates this point. In Sigma, whenever a trusted process has to take an action that would violate the security model, the user is required to press a key to authorize the process to take that action. A log is kept of these instances so that the user can be held accountable for these exemptions to the policy. Interviews with users reveal that most users did not understand why they were asked to confirm certain actions at certain times. They always pressed the confirm key when they were asked for confirmation, because that was the way to make the system do what they wanted. This practice raises serious doubts about the security of the system.

Application-Specific Policies

Another example of poor fit relates to operations specific to the application and relevant to security. In military message systems, a formal message is a communication on record that can be released only by persons with release authority. If a user of the system without that authority were to cause a message to be released without the approval of someone with such authority, it would be considered a breach of security. The conventional generic approaches do not account for such specific requirements.

Advantages of the Application-Based Approach

The approach taken to develop the SMMS security model is aimed at overcoming these problems by capturing the actual policy that the system will present to its end users. This policy is founded in the intuitive notions of security by which the application users judge the system's behavior. No trusted processes are introduced. All application programs conform to the same security policy. It is not necessary to grant exemptions because the user's intuitive notions of security form the basis for the model up front. The security model is constructed from concepts familiar to end users by describing security in terms of users performing operations on entities. The entities correspond to objects of the application such as messages or files of messages. The operations correspond to the application's user commands. The SMMS model encompasses security policies specific to the message-processing application. Operations such as message releases are explicitly dealt with. The result should be a security model that exhibits a good fit with the intuitive security requirements of the application.

Security Model Definition

Informal and Formal Security Model Definitions

The model defining the security requirements of the system is documented in Ref. 3 in an informal and a formal form. The informal description is meant to be understood by the system end users. It consists of definitions, assumptions, assertions, and a model of operations. Terms fundamental to the security model, such as user, entity, and container, are defined. The assumptions include items the system cannot enforce but that are required for the system to be secure, e.g., that users label information appropriately. The security assertions are requirements that the system can and must enforce to be secure. Finally, a model of operations puts the definitions in context by briefly describing how users interact with the system.

The formal model is derived from the informal model and is intended to be used by the system designers and certifiers. To be unambiguous, the formal model defines security with mathematical rigor in terms of set theory and predicate calculus. The arguments made about the security of the prototype are based on showing a correspondence between the design we describe here and the formal model. Although correspondence to a formal model is the subject of our discussion, most of the arguments in this document are made informally. We also rely on the informal statement of the model to provide an easily understood description of the security requirements.

Basic Terminology

To understand the analysis presented, it is necessary to be familiar with definitions from the SMMS model. This will allow the reader to understand the security analysis that follows; however the statement presented in Ref. 2 remains the official definition of the SMMS security model.

Entities are objects in the system that hold the information we want to protect. Each entity has a *classification* denoting the sensitivity of the information it holds. *Users* have *clearances* that denote the level of security granted to the user based on background and security checks. Every user is assigned a unique *user identifier*. Users are also identified with *roles* that allow extended capabilities. For example, the *system security officer* (SSO) role is required to set the clearances recorded by the system for other users. Each user is assigned a *current role set* in which the user is currently acting and an *authorized role set* in which a user is authorized to act. The current role set is always a subset of the authorized role set.

Each classification and clearance is represented by a *security level*, an ordered pair designating a *sensitivity level* (for example, TOP SECRET, SECRET, CONFIDENTIAL, UNCLASSIFIED, in strict decreasing order) and a set of *compartments*. A compartment indicates a further subdivision of information based on need to know. The security levels are ordered in a relation that may be described mathematically as a lattice [4]. We refer to this order with the term *dominates*. For example, we may say that the classification of an entity dominates the classification of another entity, or that a user's clearance dominates the classification of an entity. Sometimes we abbreviate this by saying that one entity dominates another, or that a user's clearance dominates an entity. A more precise definition of dominates may be stated: given security levels *s1* and *s2*, we say *s1 dominates s2* if and only if the sensitivity level of *s1* is at least as great as that of *s2*, and the compartments of *s1* are a superset of the compartments of *s2*. By definition, every security level dominates itself.

We will use entities to model objects familiar to message system users. These objects will include message files, messages, the fields within messages (e.g., To, From, Subject), and possibly the individual paragraphs within a message. Each entity is associated with a *value*, which may be thought of as a string (of characters or bits) representing the information to be protected. Its exact structure is not of concern to the security model.

One intuitive rule that users apply to a secure message system application dictates that an UNCLASSIFIED message should not contain a SECRET message field. To model such rules, we introduce a containment relationship between entities. To speak more precisely about containment, more terms are defined. Entities may *contain* other entities. An entity that contains another entity is called a *container*. The entities contained by an entity may themselves contain other entities. This containment relationship between entities is referred to as the *containment hierarchy*. There is no necessary correspondence between the value of an entity and what the entity contains. The value and content should be thought of as independent attributes of an entity. It is possible to make an entity contain other entities without affecting its value.

A user refers to a specific entity by providing a name, or *reference*, to the entity. An entity can be referenced directly or indirectly. As we will see later, the way an entity is referenced may affect what security checking is performed when that reference is interpreted. Each entity is associated with a *unique identifier*, a string, that in a given state, uniquely identifies that entity. Such a string might be ''MF134.'' A *direct reference* to an entity consists of just this unique identifier. When a direct reference is used to refer to an entity, security checking is independent of the entities that contain it. An *indirect reference* refers to the entity in terms of some entity that contains it; for example, ''the third entity contained in DIR77.'' When an indirect reference is used, additional security checks are made that depend on the entities along the path.

Users interact with the message system by performing *operations* on entities. An operation may have several parameter positions in which an entity may appear. To provide a discretionary mechanism for indicating who may perform what operation on what entity, we associate each entity with an *access set*. This is a set of triples of the form *(u,op,k)*, where *u* is a user identifier or role, *op* is an operation, and *k* is a parameter position of *op*. The intended interpretation of the access set is that the user *u* is permitted to use the entity in the *k*th parameter position of the operation *op*. (The access set of an entity is independent of its classification. One should not conclude that a particular triple is in an access set based on the classification of an entity and the clearance of a user. The access set should be thought of as a separate attribute of an entity on a par with its classification, content, and value.)

We want to be able to make precise statements about releasing formal messages. To distinguish between released and unreleased messages we associate each entity with an *entity type*. The entity type can be released message (RM) or draft message (DM). To provide accountability, every RM entity has a *releaser field* specifying the identifier of the user who released the message.

Characterizing Secure Behavior

To characterize secure behavior, the formal model first presents a mathematical model of behavior in general. Once this is done, a characterization of secure behavior is developed.

The formal security model describes the system as a state machine in a way that corresponds intuitively to the way a user interacts with the system. The system starts in a *state* characterized by the entities that exist, various attributes of these entities, the clearances recorded for the users, and other such information. The characterization of the system state is derived from the mental model that a user should have to rationalize the observable behavior of the system. The system moves from one state to another in response to requests issued by the user. Each *request* is an $N + 1$ – tuple ⟨*op,x1,...,xNRB* consisting of an operation *op* and a list of parameters *x1,...,xN*. The parameters may be references to entities, user identifiers, or arbitrary strings. These requests are intended to correspond to the user commands of the application software. The possible state transitions are defined by a *system transform*, a function T that takes a user identifier u, a request i, and a state s, and maps them to a new state s^*, written $T(u,i,s) = s^*$. The system transform captures the intuitive notion of how the execution of user commands (requests) affects the system state. A *history* of the system is a record of the sequence of states through which the system moves, along with the user and request responsible for each state transition. This history represents the behavior the user can observe of the system. A particular transform, along with a set of possible initial states, determines the set of possible histories allowed, which in turn characterizes all possible behaviors of the system.

The next step is to characterize secure behavior for such a system. To do that, we describe properties intended to constrain the behavior of the system to be secure in an intuitive sense. We call these properties *security properties*, or *security assertions*. Some of these properties can be described in terms of the structure of a state independent from its context of preceding or succeeding states. For example, we can determine that no UNCLASSIFIED message contains a SECRET message field by examining a single state's structure. Other security properties may require us to inspect not just individual states, but also the system transform; such properties constrain the changes in state that are allowed. For example, we can capture the intuitive requirement that only the SSO can modify clearances by the following rule. If in one state a user has a given clearance and in the following state that user has a different clearance, the user who made the request that caused the state change must have the authorized role of SSO in the starting state.

Necessity of Considering History

The technique of making assertions over histories instead of individual states is a departure from conventional security models whose assertions apply only to isolated states. John McLean, of NRL, has observed that the Bell-LaPadula model fails to take this necessary step and, as a result, systems can be constructed that conform to the Bell-LaPadula model that are intuitively insecure. McLean constructs a system to have this property, called System Z, which includes an operation that simultaneously lowers all the classifications and clearances of entities in the system [5]. Such an operation is intuitively insecure, but does not violate any of the security assertions that the standard formulation of the Bell-LaPadula model [1] places on states. The states before and after the operation are secure under that model's definition of a secure state. The SMMS formal security model avoids this mistake by requiring the system transform to be secure.

Security Assertions

Having covered the basic definitions and issues in devising a security model, we move to the specific assertions that make up the security model for the SMMS. In the formal presentation of the model, the properties we introduce are formalized by definitions of a secure state and a secure transform. The system is defined to be secure if all of the states in all of its histories are secure and if its transform is transform secure. State secure and transform secure are then defined as the conjunction of a number of assertions. If the system fails to conform to any one of these assertions, it is considered insecure. This suggests we can argue the security of the system by taking the assertions one at a time and showing how each assertion is enforced by the system. We will follow this basic organization.

Assertions in the formal model correspond to assertions in the informal presentation [2]. We describe our arguments in terms of these informal assertions rather than presenting all the details of the mathematical structures in which they are formalized. This approach prevents us from becoming mired in details before we have laid out the broad framework as an informal proof. This informal proof gives us a good starting point for refining both our design (by writing detailed module specifications) and, subsequently, our security arguments (by constructing a more rigorous mathematical proof).

For the system to be considered secure, all of the following assertions must hold:

Classification Hierarchy—The classification of any container always dominates the classifications of the entities it contains.

Access Secure—A user can invoke an operation on an entity only if the user's user identifier or current role appears in the entity's access set along with that operation and with an index corresponding to the operand position in which the entity is referred in the requested operation.

Labeling Requirement—Any entity viewed by a user must be labeled with its classification.

Viewing Requirement—A user can view (on some output medium) only an entity with a classification less than or equal to the user's clearance and the classification of the output medium. (This assertion applies to entities referred to either directly or indirectly).

Container Clearance Required (CCR) Secure—A user can have access to an indirectly referenced entity within a container marked ''Container Clearance Required'' only if the user's clearance dominates the classification of that container.

Copy Secure—Information removed from an object inherits the classification of that object. Information inserted into an object must not have a classification higher than the classification of that object.

Translation Secure—A user can obtain the unique identifier for an entity that the user has referred to indirectly only if the user is authorized to view that entity by that reference.

Set Secure—Only a user with the role of SSO can set the clearance and role set recorded for a user, or the classification assigned to a device. A user's current role set can be altered only by that user or by a user with the role of SSO.

Downgrade Secure—No classification marking can be downgraded except by a user with the role of downgrader who has invoked a downgrade operation.

Release Secure—No draft message can be released except by a user with the role of releaser. The user identifier of the releaser must be recorded in the releaser field of the draft message.

Experience with the Security Model

Earlier prototype systems [2,6] implement the SMMS security model for the purpose of investigating the interactions between that model and a specific set of functional requirements [7]. Based on our experience with these prototypes, we conclude that the SMMS security model is useful, that it is understandable by users, and that it does a good job of capturing the intuitive notion of security the message system users apply to the information they handle. Our experience with prototypes implementing the security model is that intuitively secure operations are allowed, and intuitively insecure operations are disallowed. The SMMS security model overcomes the problems of a poor fit between the security model of the computer system and the security needs of the application identified by earlier projects, such as the Military Message Experiment [8]. We believe we can attribute these properties to the application-based approach we used to derive the model.

The application-based approach appears sufficiently general to apply to a wide range of applications with good effects. Do not interpret the work presented here as applicable only to secure message systems. Modeling the security-relevant behavior of the system as a state machine and describing security properties in terms of histories is a very general approach. It is arguably more general than approaches that take a conventional model as their starting point and proceed by introducing trusted processes.

GOALS OF THE SECURITY ARCHITECTURE

Whereas the security model must capture a set of requirements, the security architecture must capture a set of design decisions. These design decisions are constrained not only by the security requirements but by the requirements of the system as a whole. To guide us in making these design decisions and evaluating the resulting architecture, we need to define goals.

The primary goal of the security architecture relates to the security of the system:

- The architecture must enable us to provide strong assurance that the system behavior will be secure as defined by the SMMS security model. The architecture must result in a design amenable to proof of its security properties by manual or automated means.

The architecture must also meet additional constraints imposed by the requirements for a family of military message systems:

- The architecture must apply to a wide range of hardware and software bases. Some of these bases may require the application to interface with specific operating systems, machine instruction sets, and devices.

- The architecture must apply to a message system distributed across workstations connected by a local area network as well as terminals connected to a central processing unit (CPU), or a stand-alone workstation.

Note that these goals imply tradeoffs. Strong assurance implies that the system should be kept small and simple. The requirements of a family of systems dictate that the system should be factored into a number of components to encapsulate the differences between systems in the family. The additional interfaces required to do this complicate the design and result in a larger system. To the extent the size of the system is increased, the assurance task becomes more difficult.

Fortunately, the goals are complementary in an important respect. Both the construction of a software family and the attainment of high security assurance require careful software engineering. The intermediate products of a disciplined approach (such as a module decomposition and precise module specifications) are doubly useful because they are essential to addressing both concerns. Because the payoff is so great, the SMMS prototypes rely greatly on a software design methodology that emphasizes modular decomposition and precise specification of module interfaces.

SECURITY ARCHITECTURE DECOMPOSITION

The architecture is implemented as a set of software structures operating on top of a lower level construct, the *abstract base machine* (ABM). The ABM provides the primitive security functions for the implementation of higher level structures required to implement the model. These functions support protection structures called *domains*. The structure on top of the ABM uses this domain-building capability to construct a specific set of domains that interact according to strict rules. This specific set of domains and the rules that govern their interaction form the *domain structure* of the system. Within this domain structure, the *entity monitor domain* carries most of the system's security responsibilities.

The design decisions given in this report (i.e., the properties required of domains, the specific domains chosen, and the rules given for their interaction) allow us to construct a system for which we can provide sound arguments that the SMMS security policy is enforced. To verify this informally, we analyze the software structure defined here in terms of the assertions of the security model. For each assertion, we describe how the design decisions made in defining the architecture enable us to build simple mechanisms that guarantee the validity of that assertion.

THE ABSTRACT BASE MACHINE

Motivation

Since the SMMS must be implementable on a variety of hardware and software bases, we are not free to choose a particular base machine and use its characteristics to construct our implementation. We do, however, enumerate a set of characteristics that are required of the base machine and then assume these characteristics. We describe an ABM that may be implemented on top of an existing operating system or bare hardware. This ABM approach is the primary way of attaining independence from any specific operating system or hardware.

Basic Protection Mechanisms

To make the job of providing strong security arguments feasible, the ABM must support basic services such as reliably separating data and mediating access to that data.* The security design will rely on the ABM's ability to provide protection mechanisms with sufficient assurance of the classical properties of a reference monitor.

1) Completeness—all accesses are checked.

2) Correctness—the mechanism performs as specified.

3) Inviolability—the mechanism cannot be tampered with.

*The ABM is very close in functionality to the separation kernel proposed by Rushby [9]. In Rushby's terminology, its main concerns are *separation* and *mediation*.

We can use the protection mechanisms of the underlying machine or operating system to construct the *protection mechanisms* of the ABM. Assurance that the above properties hold for the ABM relies on the guarantee that they hold for the underlying machine.

Protection mechanisms can be modeled by an access matrix model similar to that described by Lampson [10]. The idea is that programs are limited in what they are permitted to access by the context in which they execute. Lampson uses the term *domain* to describe this context. We define a domain as a named set of access permissions. Whether or not an executing program can perform a particular access is determined by the domain with which it is associated.

System Objects

The ABM provides a set of *system objects* (e.g., files, segments, ports, message queues). All programs and data reside in system objects. An access matrix associated with each domain determines whether or not a system object can be read, written, or executed in that domain. The ABM provides mechanisms to define processes that execute code stored in system objects to alter other system objects. This is the only way that system objects can be altered. A process is associated with a single domain for its lifetime, although the access matrix of a process' domain may change. A process may indirectly access system objects in another domain by creating or communicating with another process that executes in a different domain.

Conceptual Tools for Assurance

We want the ABM to provide structures that will help us obtain security assurance by reasoning about our design. The notion of invariance is an important conceptual tool in this reasoning. The ABM's provision for domains directly supports this reasoning tool.

Invariant properties are properties that are true at all times. For example, i may be an integer counter represented in the system, and a property may be that i is positive. This property may be expressed with the predicate $i > 0$. We assume that the system is made up of a set of programs that may be executed repeatedly in any order. If no possible order of executing these programs can leave $i > 0$ false, then $i > 0$ is *invariant.* To prove that $i > 0$ is invariant, it would suffice to show that $i > 0$ is true in the initial state of the system and that for each program that modified i, if $i > 0$ held before the program executed, then after execution $i > 0$ would necessarily hold.

This proof technique is general and can be extended to more interesting predicates. It can be applied to predicates that express the security requirements of the system. Recall that the secure behavior of the system is characterized by a set of security assertions. These security assertions can be stated as invariant properties of the system state. We call these *security invariants.* These security invariants can be shown to hold for the system by the same proof technique we apply to show other properties invariant.

Notice that in the example above, the proof technique requires us to inspect all of the programs that could modify i. In a large system with thousands of programs, our task would be difficult indeed if some mechanism were not present for limiting which programs have access to i. We would have to inspect every program in the system as well as every new program added to the system over its lifetime. If we can limit access to i to just a small number of programs, invariant properties of i can be verified by inspecting just those programs that access i. The domain is the structure that allows us to limit such accesses.

ABM Support for Conceptual Tools

The ABM facilities make it possible to construct mechanisms that isolate system objects in separate domains and control access to those objects in a flexible way. For example, sensitive information S can be represented in a set of system objects for which no domain other than A allows access. Domain A may include a set of code objects that are executable only by domain B. Thus processes running in domain A have direct access to S, processes running in domain B may have indirect access to S (called "indirect," since it is only by executing code objects in A that S can be accessed), and other processes have no access at all. We can prove invariant properties about S simply by inspecting the executable system objects in A. Call the system objects in A executable by B, *gates* from B to A. We can enforce a more specific policy for B's access to S in the gates to A. This technique is used in the SMMS design to minimize the security argument and increase security assurance. It is implemented in the application layer built on top of the ABM.

THE DOMAIN STRUCTURE

Design Strategy

On top of the ABM, we implement the message system application by dividing its code and data among system objects in different domains. The domain structure is chosen to partition out the security responsibilities to different domains. Domains will normally be associated with a set of invariant properties that are required of the data residing in that domain. These properties are predicates over the data stored in system objects. It should be possible to verify that these properties hold by inspecting only the programs that may access data in that domain.

In the discussion that follows, we examine the kinds of security properties we need to enforce and see how they suggest what information might be placed in specific domains to enforce these properties.

Motivations for Selecting Specific Domains

Security Model as Basis

The choice of the specific domains is motivated by the form of the security assertions that make up the SMMS model. The form of these security assertions implies some of the security responsibilities that parts of the system must have. For example, many of the formal security assertions given in Ref. 2 are of the form:

$$\text{For all } u,i,s,s^* \text{ such that } T(u,i,s) = s^*, P(u,i,s,s^*),$$

which means that for all users u, requests i, states s,s^* (such that user u issuing request i in state s transforms the system to state s^*), a particular property P must hold over u, i, s, and s^*. To enforce such assertions, it is necessary that the system be able to authenticate users so that it can associate their identity with the requests they enter. Thus authentication of users and association of users with requests become responsibilities we would like to be able to associate with some set of domains.

Need to Associate Actions with Users

We can associate users with the requests entered by associating the input device with a particular user for a period of time. Users normally interact with the system by first presenting a user identifier and password (or some similar authentication) over an input device such as a terminal keyboard.

From the time the user is authenticated to the time that user logs off, it is assumed that the input from the terminal correctly reflects the actions (key presses, menu picks, etc.) of that user.

Invariant Properties on System State

Other security assertions are concerned with invariant properties of the system state. For example, the classification hierarchy assertion says that the classification of an entity must dominate the classifications of any entities it contains. This and similar assertions can be enforced by isolating, in a particular domain, the system objects that make up the state. These objects will be modified only by using gates that perform the checking necessary to maintain those security assertions.

Isolation of User Requests

A separate domain for executing user requests enables us to isolate the system objects affected by a request. Provision for such a domain is motivated by efficiency considerations as well as ease of proof.

We want to be able to ensure that all of the objects an operation can access satisfy some particular properties before allowing the program that will carry out that request to run. If the operation could potentially access all the objects in the system, performing such a check could have disastrous results on performance. Performing the necessary checks becomes feasible when we have to inspect only a small number of objects. For this reason, we set up a separate domain and place the operation, and the objects it will access, in that domain before allowing the operation to run. This way, we only have to ensure that the objects we place in that domain have the necessary properties. We are guaranteed that the executing operation cannot directly access objects outside that domain. This allows us to check certain properties at the time the objects are installed and allows the operations to run within the domain unfettered by additional checking.

Ease of proof is also a benefit of this approach. The mechanism that installs objects in the domains is a small, well-defined part of the system whose correct operation implies a great deal about the overall security of the system. Many security properties may be proven by inspecting this small part without the need to make complex assumptions about the other parts. Since a small component is usually easier to verify than a large one, this makes the proof task more tractable.

Control Over Input and Output Devices

Finally, other security assertions control the output that the system presents to users. Like input, output appears on devices that are associated with users. These output devices often have special characteristics that are potential security problems. For example, it is often possible to use the special characteristics of a terminal to masquerade as the user logged in on that terminal. On most systems, between the time a user logs in on a particular terminal and the time that user logs out, the system treats all characters it receives from the terminal as representing the actions of the user who logged in. Now consider that many terminals recognize escape sequences that are interpreted to mean ``transmit everything displayed on line *i* through line *j* of the screen as if it were keyed in on the keyboard.'' We wouldn't want to permit an attacker to embed these escape sequences in a message so that when the message is displayed at the victim's terminal, it appears to the system as if the victim has keyed a sequence of characters designated by the attacker. Enforcement of the SMMS security model relies on the strict association of users with their actions.

Isolating output devices in separate domains helps us overcome these sorts of problems. Programs will be able to interact only with the output device through gates to the domain that isolate the output device. The programs that implement these gates will be responsible for taking into account special characteristics of the device, such as the one just described.

Choosing the Specific Domains

This discussion of security responsibilities suggests the following five types of domains.

Input Server Domain (one per input device)

Each input device is identified with and isolated in a unique input server domain. Programs executing in this domain authenticate users and send user input to the client domain associated with that user, once authenticated.

Client Domain (one per user)

A process in each client domain acts as the agent for a user. Each client domain is associated with a particular user and serves to isolate that user. A program executing in this domain must correctly translate input it receives from a particular device domain into requests.

Entity Monitor Domain (one per SMMS)

The entity monitor domain isolates the SMMS state and maintains security invariants on it. The security invariants are derived from the predicates on system histories as defined in the security model.

Invoke Domain (one per user)

Operations identified by user requests execute only in the invoke domain. This is the only way user operations can modify the system state. To limit the access executing operations have to the system state, only those portions of the system state that the operation needs to access are made available in this domain.

Output Server Domain (one per output device)

Each output device is identified with and isolated in a unique output server domain. To produce output, programs must communicate with the output server programs that access the output device.

Figure 1 shows how these domains are interconnected.

PROCESSING WITHIN DOMAINS

To understand how these domains interact, it is helpful to follow the processing of a user command as it is entered on an input device and as it accesses the system database and generates output on an output device.

Input Server Domain Processing

The user's input originates at a device isolated in the input server domain. Access to this input device is regulated by controlling access to its input server domain, which is responsible for authenticating the user and routing that user's input to the correct client domain for that user. Once the user has been authenticated, the user's input is sent to the client domain (and only the client domain) until the session is terminated. When the session is terminated (e.g., by logging off), the input server stops routing input to the client domain and waits for another user to be successfully identified.

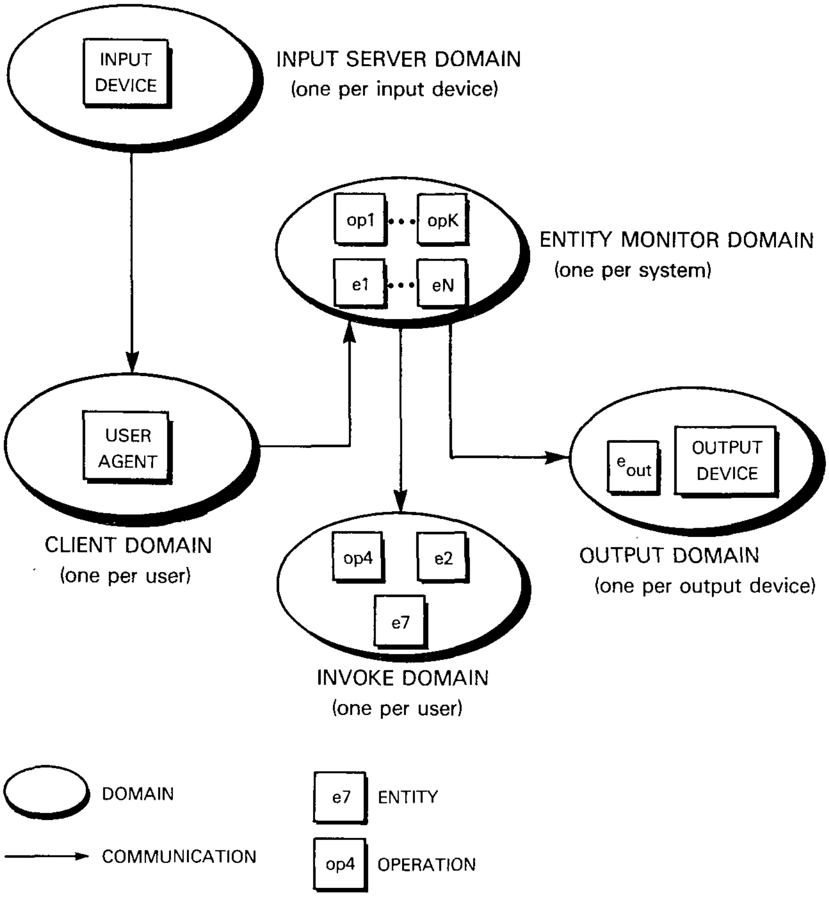

Fig. 1 — SMMS domain structure

Client Domain Processing

During the session, the user's input passes from the input server domain, which identifies the input device, to the client domain associated with the authenticated user. Programs running in the client domain trust the input server to have correctly identified the user and assume that the data they receive from the input server represents *exactly* the actions taken by the user (e.g., the keys typed, menu items picked, mouse motions made). All actions taken by the client domain are considered to be at the user's command, so that communications from the client domain ultimately may be associated with the user.

A process in the client domain translates the information it receives from the input server into requests consisting of an operation and a number of parameters. The operations are those defined in the Intermediate Command Language (ICL) for the SMMS family [7]. The programs running in the client domain are trusted to perform this translation, correctly representing the user's requested operation. These requests are passed to programs running in the entity monitor domain.

Entity Monitor Domain Processing

The ABM provides a security mechanism that protects system objects. At the higher level, the entity monitor provides the remainder of the system with a policy centered around protecting *entities*. The entity monitor can be viewed as a *protected subsystem* [11] constructed from system objects using the ABM mechanisms to provide its clients a more application-specific set of mechanisms to protect entities.

A set of system objects collectively represents the *secure state* defined by the SMMS security model. All system objects used to represent the SMMS state reside in the entity monitor domain. All requests to perform operations on the information represented in the SMMS state pass from the client domain to the entity monitor domain. A request includes an operation name (specifying an executable system object) and a list of system object names denoting the *entities* to which the operation is to be applied. The gates to the entity monitor domain perform the checks necessary to enforce the SMMS security model. For example, the access sets of each of the operands of the request are checked to determine that the operation and user are in the access set for the operation. If the request passes those checks, then the system objects representing the operation code and the operands are placed in the invoke domain. The programs in the entity monitor domain are required to perform these checks correctly and to place only the operation and operands listed in the request in the invoke domain.

Invoke Domain Processing

In the invoke domain, the operation executes in isolation from the rest of the system. The process running the operation code can directly access only the system objects that have been placed in the invoke domain by the entity monitor. Moreover, it allows us to set up the read/write permissions on these operands so that information flow is prevented from objects with higher classifications to objects with lower classifications. For example, if a both a SECRET object and an UNCLASSIFIED object are being installed in the invoke domain, we can guarantee that if the domain allows read access to the SECRET object, it does not allow write access to the UNCLASSIFIED object. This would prevent, for example, the use of a SECRET file to modify an UNCLASSIFIED file.

As the operation code executes, the invoke domain processes may use gates back to the entity monitor to access indirectly other entities. The entity monitor prevents operations from being performed on any entities that were not installed in the invoke domain. It is possible for an operation to request that entities be installed that were not installed initially. For example, if an operand is a message-file, the operation may want to inspect the messages contained in that message-file. Since the messages are represented in system objects separate from the message-file, the operation must ask the entity monitor to install the messages in the invoke domain before the operation can access them. The entity monitor will check the access sets and read/write permissions at the time it installs the requested entities, just as it did for installing the initial operands.

Output Server Domain Processing

System objects that correspond to output devices are placed in the output server domain. Requests to display entities on output devices are made through gates to the output server domain. The output server domain is set up to guarantee that no part of the system can access any output device except by communicating with an output server.

Each output server recognizes a special entity maintained by the entity monitor as a structure that holds the information to display to the user on a particular output device. Taking advantage of what they know about the particular characteristics of the output device to provide an appropriate representation, the programs in the output server have the responsibility of depicting that information on the output device in a way that is understandable to the user. For example, the special entity may be a message. In this case, the output server translates the information in that entity to a representation of the message to display to the user. Ultimately this results in a sequence of characters (including control information) transmitted to the output device. Operations cannot send data directly to the output devices; operations can affect the output devices only by modifying the special entities that the output server programs watch and continuously interpret. This prevents programs from taking advantage of any escape sequences or other control sequences that might be used to manipulate output devices with undesirable effects. Thus the results of the user command given to the input server will be seen only on an output device if the operation running in the invoke domain causes a change to the special entity associated with that output device.

THE ENTITY MONITOR

The SMMS security model defines a number of abstract data types and operations on those data types. The data types include entity, security level, and access set. Our approach to construct a secure system based on this model is to implement programs that support the necessary manipulations of these data types. These access programs allow client software to create entities, reclassify entities, and make containment relationships between entities. The collection of programs that implement these functions makes up the entity monitor (EM). These programs execute only in the EM domain. The most important characteristic of the EM is that it enforces security assertions described in the SMMS security model.

The ICL operations that manipulate entities can do so only indirectly by using the EM interface. The degree to which the operations can affect the system state thus becomes constrained by the invariants the EM maintains on that state. For example, a program that inserts a citation into a message file must use the EM program to make a containment relationship between the two entities. If the operation attempts to insert a SECRET entity into an UNCLASSIFIED one, the EM will prevent the insertion from being made. The operation can detect this event and provide an appropriate response.

The existence of a correct set of EM programs is not sufficient to guarantee the security of the system. It is still necessary that its client programs use the EM properly. This can be illustrated with an example of a misuse of the EM. Assume a message file is interpreted as an entity containing messages that are themselves entities. The security policy dictates that an UNCLASSIFIED message file never contains a SECRET message. However, if the programmer decides not to represent message files as entities nor messages as the entities contained by message files, the EM could not guarantee the security of the resulting implementation. A misguided programmer might choose to represent this information by encoding everything in the value of a single EM entity. This is analogous to a programmer implementing his own miniature file system in a single operating system file instead of allocating separate files. Code review, or a similar procedure, must be used to ensure that programmers are actually placing the information to be protected in the EM structures where the security policy is enforced.

SECURITY PROPERTY ANALYSIS

The remainder of this report analyzes the security architecture in terms of the assertions of the SMMS security model and describes how these assertions are maintained. The goal of the sections that follow is to indicate clearly what mechanisms in the design are responsible for maintaining these assertions and to provide the rationale for asserting that the architecture resulting from this set of design decisions can, in practice, be made secure in the sense of the SMMS security model.

Based on its informal description, we take each of the security assertions in turn and show how we could guarantee that the assertion holds as the result of the design decisions made in this report. At the same time, we may present specific new design decisions necessary for an informal proof of the assertion. Among these design decisions are gates to specific domains and checks that these gates must perform. For each security assertion, we attempt to provide the information needed to enforce the assertion, to identify the mechanisms responsible, and to show that the necessary checks could be constructed.

We do not attempt a rigorous definition of the semantics of the gates and their security checking or provide a rigorous proof of their sufficiency. These details are voluminous and are left to the module specification and security proof documents. Also, doing the job right requires more mathematical rigor than is appropriate to this report. Our aim here is to provide a high-level outline of the design decisions and security arguments so the reader will understand their overall structures and gain an intuitive feel for how the design decisions result in a system that satisfies the security requirements. Then the reader should be prepared to tackle the more rigorous treatments in the security proof documentation.

Classification Hierarchy

The classification of any container always dominates the classifications of the entities it contains.

The classification hierarchy assertion can be viewed as a predicate on the system state. The predicate mentions only entities and three attributes of entities. The attributes are the existence of, classification of, and containment relationship among entities. For the EM to enforce this assertion, it must mediate any changes to these attributes of entities. All of this information is represented in data objects in the domain of the EM. Table 1 lists the EM gates that are used to access this information. All the tables in this report are intended to illustrate the accesses that operators define by the EM design. The actual design may include more or slightly different parameters and have more complex semantics, but for the purpose of security analysis, it is essentially the same. For a definitive description of the EM interface, the reader is directed to the module specifications [12].

Table 1 — Access Operators for a Simple Entity Monitor

Gate	Description
NewEnt(level)	Creates a new entity at the given security *level* and returns that entity. Establishes the condition *ExistsEnt(NewEnt(level))=true* *& Classif(NewEnt(level))=level* *& ExistsEnt(SubEnt(NewEnt(level),i))=false for all i.*
SetSubEnt(ent1,i,ent2)	Makes a containment relation between *ent*1 and *ent*2 such that *ent*2 is the *i*th entity contained by *ent*1 if and only if *ent*1 dominates *ent*2. Establishes the condition If *Classif(ent1)* ≥ *Classif(ent2)* then *SetSubEnt(ent1,i)=ent2*
ExistsEnt(ent)	Returns true if and only if an entity with the name *ent* exists. It has no effect on the system state. In the initial state, it returns *false* for all ent.
SubEnt(ent1,i)	Returns the *i*th entity contained by *ent*1; returns *undefined* if *ent1* contains no such entity. It has no effect on the system state.
Classif (ent)	Returns the classification of *ent*. It has no effect on the system state.

It is a simple matter to implement the operations of Table 1 as programs in the ABM architecture we have described. Each entity is represented by two system objects. The first (called an access controller) always resides in the EM domain. The second holds the entity's value and is described with other security assertions later in this report. We focus now on the access controller, where we store the entity's security level, a list of any entities it contains, and a pointer to a system object that holds the entity's value.

When *NewEnt(level)* is performed, the EM must allocate two system objects, one for the access controller and one for the entity value. The access controller is initialized with the given security level and the name of the system object representing the entity value. The value is initialized to empty. The EM returns the name of this access controller to the caller.

SetSubEnt(ent1,i,ent2) is implemented by having the EM trace along the containment links in the access controllers and modify that list to make *ent2* the *i*th entity contained in *ent1*. As it does this, it checks to ensure that the hierarchy assertion is not violated.

SubEnt(ent1,i) is similar to the above two operations. It must follow the links in the access controllers to find the *i*th entity in *ent1*. It returns the name of that access controller to the caller.

It should be clear that implementing the EM with the above representation presents no major problems. Client programs may refer to entities by the names of their access controllers, but they cannot access what is stored in these system objects except indirectly through the EM. For example, when a client program declares a variable of type entity, what that variable will hold is the name of an access controller, not the information stored in the access controller.

By inspection, the truth of the hierarchy assertion depends on just three attributes of entities: existence, classification, and the containment relationship among them. These three attributes are under complete control of the entity monitor, since they are represented in the access controllers directly accessible only by the EM. We assume that when the system is initialized, it is put in a secure state so that the classification hierarchy assertion holds initially. The existence, classification, and containment relationships among entities will only change as the result of the effects of the gates in Table 1. It remains to examine the effects of those gates to determine whether or not they alter the state in such a way that classification hierarchy no longer holds. We can restate classification hierarchy as a predicate on the system state using the value returning gates of the module interface.

$$(Hier) \quad ExistsEnt(SubEnt(ent,i)) \rightarrow Classif(ent) \geq Classif(SubEnt(ent,i)) \text{ for all } ent,i.$$

Having done this, we show that this predicate is invariant over each gate of the EM interface. *Exists Ent*, *SubEnt*, and *Classif* do not alter the state, so Hier is invariant over them. *NewEnt* creates entities that contain no other entities, so Hier is invariant over them as well. *SetSubEnt(ent1,i,ent2)* would have the potential to invalidate Hier if it set *SubEnt(ent1,i)* to the value of *ent2* when executed in a state where *Classif(ent1)* ≥ *Classif(ent2)* is not true. Thus we have defined *SetSubEnt* to check for this condition and not alter the state when that condition is detected. No other EM gates affect Hier. Hence, Hier is invariant.

This proof, typical of the proofs for the SMMS, depends on the absence of other gates that may be used to modify the specific attributes we are considering. The proof would be made invalid if we introduced additional EM gates that modify an entity's existence, classification, or containment relationship. If we introduce such a gate, we will have to show that Hier is invariant over that gate as well in order to maintain our proof. The proof does not depend on the addition of programs in domains other than the EM domain. Adding programs to those domains does not require any maintenance on this proof.

The proof depends on an implicit assumption that the state does not change spontaneously if no EM gate is executed. This property is part of the specification for the EM programs. This assumption is actually stronger than we need for the proof but is made out of consideration for other services that the module should provide to its users. (For example, classification hierarchy would still hold if unclassified entities spontaneously became contained by other entities, but such behavior should cause problems to a programmer trying to implement a message file in terms of entities.) Weaker assumptions might be substituted to accommodate failure modes of the system. For example, a disk crash might cause entities to disappear apparently spontaneously without calling EM gates. The proof will still go through on the assumption that entities may spontaneously disappear, but not spontaneously appear.

In summary, we have shown that if the EM operations are correctly implemented, if all other operations in the system do not affect these specific attributes of entities, and if the system begins in a secure state, then the classification hierarchy predicate is satisfied over all system histories. This report applies the same form of argument to the remaining assertions of the SMMS security model.

Access Secure

> *A user can invoke an operation on an entity only if the user's user identifier or current role appears in the entity's access set along with that operation and with an index value corresponding to the operand position in which the entity is referred in the requested operation.*

We extend the design to enforce the above assertion. The above assertion says that user *u* is allowed to apply the operation *op* with entity *e* as the *k*th operand if and only if the triple *(u,op,k)* is in the access set of *e*. To enforce this restriction, the EM must be aware of every attempt to apply an operation and be able to identify the user performing the operation, the operation, and the entities to which the operation is being applied.

The EM can identify the user by identifying which client domain sent the request. Recall that the input server is trusted to authenticate the user and to forward the user's input only to the correct client domain. Because the client domain is trusted to send requests to the EM only in response to the user's input, we can conclude that the user associated with that domain did originate the request.

All ICL operations are implemented as system objects that execute only in the invoke domain. The EM must install the operation in the invoke domain before it can execute. The EM identifies the operation to be performed by inspecting requests sent to it by client domains. Programs outside the EM are forced to call the EM every time they perform an operation. We prevent client programs from ever executing operation code without help from the EM by controlling which domains are allowed to have execute permission on the system object representing the operation. Thus, the EM can always identify what operation is performed each time it is performed.

The EM must identify what entities an operation is applied to. The EM identifies these entities much the same way it identifies the operation. References to the entities to which the operation is applied are encoded in the request sent from the client domain. The system objects representing these entities must be installed in the invoke domain by the EM before the operation (which only executes in the invoke domain) can access them. Operations may also have implicit parameters. These implicit parameters are the entities the operation accesses that are not included explicitly in the request. We make the EM aware of these parameters by forcing the operation to ask the EM to install any entity that is not mentioned in the original parameter list. We can think of this as extending the parameter list "on the fly." The same checks are performed at this time as were performed for the explicit parameters at the time they were installed. In this way, the EM can always determine what entities are parameters for each execution of an operation.

The values of entities are only accessed (read or written) by operations running in the invoke domain. Further, any entity must be installed in the invoke domain before its value can be accessed. The operations that install entities in the invoke domain never install the entity unless the proper access set checks are passed. Hence, we are guaranteed that the values of entities are never accessed unless the proper access set checking has been done.

It should be clear that the EM can get the information it needs to perform access set checking. Tables 2 describes the extensions we make to the EM interface to support access sets and access set checking. These extensions provide for the definition of users, their clearances, and the roles for which they are authorized. They also provide for modification of the access sets and a mechanism for invoking operations on entities that guarantees these access sets are checked.

Table 2(a) — Users and Access Set Checking

Gate	Description
CreateUser(u)	Adds *u* to the set of users. Establishes *ExistsUser(u)=true*.
Exists(user(u)	Returns true if and only user *u* exists, false otherwise. No effect on system state.
SetUserClearance(u,l)	Sets of clearance of user *u* to *l*. Establishes *Clearance*(*u*)=*l*.
Clearance(u)	Returns the clearance of user *u*. No effect on system state.
SetActiveRoles(*u*,*roles*)	Sets the active roles of *u* to *roles*. Establishes *ActiveRoles*(*u*)=*roles*.
ActiveRoles(*u*)	Returns the active roles of *u*. No effect on system state.
SetAuthRoles(*u*,*roles*)	Sets the authorized roles of *u* to *roles*. Establishes *AuthRoles*(*u*)=*roles*.
AuthRoles(*u*)	Returns the authorized roles of *u*. No effect on system state.
SetAccessSet(*ent*,*acs*)	Sets the access set of *ent* to *acs*. Establishes *AccessSet*(*ent*)=*acs*.
AccessSet(*ent*)	Returns the access set of *ent*. No effect on system state.

Table 2(b) — Invoking Operations on Entities

Gate	Description
CurrentUser	Returns the user on whose behalf the client program is currently running.
InstallOp (*op*)	Installs the operation in the invoke domain of *CurrentUser* Establishes *OP* =*op*.
Op	Returns the operation installed in the invoke domain of *CurrentUser*.
InstallArg (*i*,*r*,*p*)	Installs the entity referred to by the reference *r* in the invoke domain of *CurrentUser* as the *i*th argument setting its read/write premissions to *p*. Establishes *Ref(i)* = *r*, *Arg(i)* = *the entity rrefersto*, *Permits(i)* = *p*.
Ref (*i*)	Returns the reference *r* to the entity last installed as the *i*th argument by *InstallArg*.
Arg (*i*)	Returns the entity last installed as the *i*th argument by *InstallArg*.
Permits (*i*)	Returns the read/write permissions of the *i*th parameter as last set by *InstallArg*.
Invoke	Executes the client operation returned by *Op* in the invoke domain of *CurrentUser*. Establishes the result of executing the operation in the invoke domain. When invoke finishes, nothing is left installed in the invoke domain.
SetValueOfArg (*i*,*buff*)	Sets the value of the entity installed as the *i*th argument in the invoke domain of *CurrentUser* to the value stored in the buffer *buff*. Establishes *ValueOfArg* (*i*)=*buff*
ValueOfArg (*i*)	Returns a buffer containing a copy of the value of entity installed as the *i*th argument in the invoke domain of *CurrentUser*.

The protocol for invoking operations on an entity is to call *InstallOp* to install the operation and to make a series of calls to *InstallArg* to install each of the arguments. *InstallArg(i,r,p)* accepts an entity reference *r* as a parameter that it interprets to determine the entity to be installed as the *i*th argument. *InstallArg* installs the entity only if *(u,op,i)* is in the access set of the entity referred to by *r*. Similarly, *InstallOp(op)* will install the operation only if *(u,op,i)* is in the access set of all of the arguments installed at the time it is called. After the operation and arguments are installed, *Invoke* must be called to actually execute the operation. *Invoke* causes the operation to execute in the invoke domain and, when the operation finishes, leaves nothing in the invoke domain.

We have described the extensions to the interface of the EM; now we describe what to add to the implementation to support these extensions. These additions are straightforward and pose no large problems. The representation of entities with access controller and value parts still suffices. We represent the access sets in the access controllers, which are under the control of the EM and can be directly accessed only by the EM.

We want to make sure that *Invoke* is the only mechanism available outside the EM for performing operations on entities. To perform an operation on an entity, a process must have the system object for the operation and the system objects for the entities on which it will operate in its domain. Otherwise the process cannot execute the operation code or access (read, write, or execute) the parameters. We make *Invoke* the only mechanism for performing operations on entities by making it the only means by which processes can execute operations. All operations are set up so that they can be executed only in the invoke domain. *InstallArg* is the only gate that installs entities in the invoke domain. When the operation finishes, invoke removes all the entities and the operation from the invoke domain before servicing the next request. Thus we are guaranteed that the only way an operation can be executed is in response to a request sent to the EM interface by the protocol described.

Labeling Requirement

*Any entity viewed by a user must be labeled with its classification.**

Users can view entities only by output devices. The EM represents output devices as special kinds of entities that allow information to be transmitted to users. Output servers executing in the Output Server Domain recognize these special entities as containers for holding information to display to the user on a particular output device. The programs in the Output Server Domain have the responsibility to display correctly that information on the output device and in a way that is understandable to the user. The EM creates the special entities representing output devices by the *Make Dev* operation. The EM can determine whether or not a given entity is a device by the *IsDevice* operation. These operations are described in Table 3.

Table 3 — Creating Output Devices

Gate	Description
MakeDev(*ent*)	Makes *ent* a special entity that the EM recognizes as an output device. *ent* must contain no other entities. *CurrentUser* must have SSO in its *ActiveRoles*. Establishes *IsDevice*(*ent*)=*true*.
IsDevice(*ent*)	Returns true if and only if *ent* is a special entity representing an output device. In the initial state, returns false for all entities.

Entities representing output devices have the same structures as other entities. They are represented by two system objects—an access controller and a value. Encoding the status of an entity, i.e., whether or not it is a device, can be implemented by including a bit within the access controller signifying the entity's status. When an entity is initially created, it is assumed not to be a device and the status bit is off. The *MakeDev* operation turns the bit on; *IsDevice* returns true if and only if the bit is on. Only the EM has the power to access these special entities and modify their device status.

Displaying entities on an output device requires calling the *SetSubEnt* operation described in Table 1. The output server associated with an output device is then responsible for ensuring that entities contained by the special entity are displayed properly. The Labeling Requirement states that

*We assume that every entity copied to an output device may be viewed by some user. Thus, we require that every entity copied to a device be labeled with its classification. Note that, although this assumption is sufficient to meet the Labeling Requirement, it may be stronger than necessary, since an output device may exist that does not allow any user to view its output.

whenever an entity is displayed on an output device, the entity must be labeled with its classification. The EM is responsible for ensuring that when a user attempts to copy an entity *ent1*, by *SetSubEnt*, in a special entity representing an output device, its classification, *Classif(ent1)*, is also contained in that special entity. Since no process outside the EM Domain has write access to output devices, no other process can interfere with the EM meeting this responsibility.

Viewing Requirement

> *A user can view (on some output medium) only those entities with a classification less than or equal to the user's clearance and the classification of the output medium. (This assertion applies to entities referred to either directly or indirectly).**

Displaying entities was discussed previously in the section dealing with the Labeling Requirement. Output devices are treated by the EM as containers. A user can copy to some output medium only those entities that are subentities of that user's output device. An entity may become a subentity only by the *SetSubEnt* operation. The classification of the entity associated with each output device, accessible through the *Classif* function, represents the classification of that output device. Therefore, those mechanisms that enforce the Classification Hierarchy requirement, as discussed previously, also enforce the requirement that the user may copy to an output device only those entities with a classification dominated by the classification of that output device.

For each request by a user to view an entity, the EM is also responsible for enforcing that the user's clearance must dominate the classification of the entity to be displayed. This check is fairly straightforward; it requires the *CurrentTerminal* command shown in Table 4. Whenever *CurrentUser* tries to perform a *SetSubEnt* on an entity *ent* for which *CurrentTerminal* = *ent*, the EM must check to make sure that *Clearance(CurrentUser)* dominates *Classif(ent)*. Since *SetSubEnt* is the only way for an entity to be displayed, this requirement must hold.

Table 4 — Accessing the Current Terminal

Gate	Description
CurrentTerminal	Returns the terminal, a device represented by a special entity, on which *CurrentUser* is currently logged in.

CCR Secure

> *A user can have access to an indirectly referenced entity within a container marked "Container Clearance Required" only if the user's clearance dominates the classification of that container.*

*As in the section on the Labeling Requirement, we assume that every entity copied to an output device may be viewed by some user. Thus, the Viewing Requirement states that (1) every entity copied to an output device have a classification less than or equal to the classification of the output device. We also assume that a user can cause an entity to be displayed, and thus potentially viewed by him, only on the terminal (output device) on which he is currently logged in. Every output device that allows viewing is considered a terminal on which some user may log in. Therefore, the Viewing Requirement states that (2) a user can view, i.e., display on his terminal, only those entities with a classification less than or equal to the user's clearance. It is the responsibility of each user to protect the entities copied to the terminal on which he is logged from being viewed by other users in violation of the Viewing Requirement. Therefore, (1) and (2) are sufficient to satisfy this requirement. For example, note that the above interpretation requires a printer to have a user logged in before any entity can be printed.

The SMMS model allows entities to be marked CCR. A user can have access to an indirectly referenced entity within a container marked CCR only if the user's clearance dominates the classification of that container. To describe these restrictions more precisely, it is necessary to discuss what references are, how they relate to entities, and what it means for a user to "have access to" an entity.

First, we consider references. Recall that the formal model defines an indirect reference $r = \langle ent, i1, \ldots, iN \rangle$ as a sequence where the first element *ent* designates an entity to start from and the remaining elements $i1, \ldots, iN$ describe a path through the containment hierarchy to the referenced entity. A reference is said to be *based on* an entity if that entity occurs along that path. A direct reference $\langle ent \rangle$ can be considered a degenerate form of the sequence with only one element.

The formal model defines access in terms of *potential modification* and *contributing factor*. A potential modification may take place when a request is issued by a user. Intuitively, a reference x to an entity is potentially modified if one of its attributes may have been changed because of the request issued by the user. If the potentially modified attributes of x depend on the attributes of reference y, then y is called a contributing factor. The terms potential modification and contributing factor do have a more precise mathematical meaning [2]. Intuitively, "potentially modified" captures the notion of writing, and "contributing factor" captures the notion of reading.

> If a request includes an indirect reference y based on z and $CCR(z)$, and there exists a reference x that is potentially modified with y as a contributing factor, then the clearance of the user must dominate the classification of the entity z.

If after executing an operation we can, by inspecting x, learn something more about y than we knew before executing the operation, that operation has somehow provided access to y in a way related to reading it. A subtler case covered by this same definition is that if $y = x$, we could have accessed y in a way related to writing it.

We extend the EM interface once again to check access to CCR entities. Requests from the client domain may now specify entity parameters as indirect references. Security checking can now depend on the paths encoded in the request. The EM interprets these paths and checks CCR restrictions before applying any operation. Since the EM supplies the services to maintain entities, we include two operations in its interface to maintain CCR marks, as Table 5 shows.

Table 5 — CCR Marks

Gate	Description
$SetCCR(ent,b)$	Set the CCR mark of entity *ent* to b. Establishes $IsCCR(ent)=b$.
$IsCCR(ent)$	True if and only if entity *ent* is marked CCR.

To perform this check efficiently, it is necessary to identify the references x and y such that x is potentially modified with y as a contributing factor. We use some informal reasoning to generate a hypothesis that gives us a sufficient check. Let "x permits writes" mean that the value segment of the entity referred to by reference x allows writing and define "y permits reads" similarly. The hypothesis is that "x permits writes and y permits reads" is a necessary condition for "x is potentially modified with y as a contributing factor." Our reasoning for this is that the only entities we need to consider are the entities $ent1, \ldots, entK$ installed as arguments in the invoke domain. These are the only entities the operation can possibly read or write. Only the entities whose value parts permit writes can be potentially modified. Likewise, only those entities whose value parts permit reads can be contributing factors, since the only way anything can contribute information is by a read.

Assuming our hypothesis is true, we can substitute "*x* permits writes and *y* permits reads" in the security model's definition of CCR secure to generate a check sufficient to guarantee that property. To ensure that the system is CCR secure, it is sufficient to check the following condition before invoking the operation on the entities installed in the invoke domain by using references *r1,...,rK*, where *ri* is installed as the *i*th argument.

> *If ri* is based on *y*, *y* is marked CCR, and *ri* permits reads or writes, then the clearance of the user requesting the operation must dominate the classification of *y* $(1 \leq i \leq K)$.

This condition can be maintained by adding checks to the *InstallArg* gate. *InstallArg* determines the entities the references are based on and compares their classifications to the clearance of the user. The argument will not be installed if the check fails.

The information to check this condition is readily accessible to the EM. As before, the user requesting the operation is identified by the client domain that sent the request. The EM can interpret the references *ri* by tracing along the links stored in the access controllers. As it does so, it can inspect the entities on which each *ri* is based. The CCR marks are stored in the access controllers. Determining whether an entity permits reading or writing is done by use of the facilities provided by the ABM interface to inspect the access permissions of system objects.

Copy Secure

> *Information removed from an object inherits the classification of that object. Information inserted into an object must not have a classification higher than the classification of that object.*

Checking for copy secure is like that for CCR secure, but simpler. The formal model says that the system is copy secure if the following holds: if *x* is potentially modified with *y* as a contributing factor, then the classification of *x* dominates the classification of *y*. Assume that "*x* permits writes and *y* permits reads" is a necessary condition for "*x* is potentially modified with *y* as a contributing factor." We use this hypothesis again to construct a simple test that the EM can perform to ensure that the system is copy secure. Recall that entities are implemented as system objects that may permit reading or writing. *InstallArg* now checks for all the installed arguments, *r*1,...,*rn*. If *ri* permits writing and *rj* permits reading, the classification of *ri* must dominate the classification of *rj*.

Translation Secure

> *A user can obtain the unique identifier for an entity that has been referred to indirectly only if that user is authorized to view that entity with that reference.*

The system is translation secure if whenever a user obtains a direct reference to an entity he has referred to only indirectly, the user's clearance dominates the classification of every entity marked CCR on which the indirect reference is based.* This property prevents users from circumventing the CCR mechanism by obtaining direct references to entities when CCR checks prevent them from accessing those entities through indirect references.

*See the section CCR Secure for a definition of *based on*.

Enforcement of translation secure requires control over which users gain access to direct references. The EM gate *CanDisplayDirect*, described in Table 6, determines whether a given user is permitted access to the direct reference of an indirectly referenced entity. The EM has the responsibility of preventing a user *u* from obtaining a direct reference to an entity referred to by *r* unless *CanDisplayDirect(r,u) = true*.

Table 6 — Displaying Direct References

Gate	Description
CanDisplayDirect (*r*,*u*)	Returns true if and only if the user *u* is allowed to obtain a direct reference to the entity referred by *r*,*i*,*e*., if and only if *r* is not based on any entity marked CCR whose classification dominates the user's clearance.

Set Secure

> *Only a user with the role of SSO can set the clearance and authorized role set recorded for a user or the classification assigned to a device. A user's current role set can be altered only by that user or by a user with the current role of SSO.*

For the system to be *set secure*, the following must hold: if the maximum classification of any output device, the clearance recorded for any user, or the authorized role of any user changes over the course of an operation, the user performing the operation must have a current role of SSO. Also, if the current role set of any user changes, the user performing the operation must either have the current role of SSO or be the only user whose current role set changed. These constraints limit changes to user clearances and the set of authorized roles to a user acting in the role of SSO.

Since the users' authorized roles and clearances are internal to the EM, it is easy to control changes. We must add information to the EM to distinguish entities that serve as output devices and record a maximum classification for these entities. We also constrain the EM gates that allow clients to modify the authorized roles and clearances in a manner that is set secure.

Both *SetUserClearance* and *SetAuthRoles* check to see that SSO is in the current role set of *CurrentUser*. *SetActiveRoles* checks to see that either the SSO is in the current role set of *CurrentUser* or that *CurrentUser* is the same as the user whose current role set is being modified. *SetActiveRoles* and *SetAuthRoles* also have the responsibility of maintaining the invariant that the current role set of each user is a subset of the authorized role set for that user.

Downgrade Secure

> *No classification marking can be downgraded except by a user with the role of downgrader who has invoked a downgrade operation.*

The system is downgrade secure if no user can lower the classification of any entity without having a current role of *downgrader*. Notice that so far we have introduced no operation that lowers the classification of an entity, so the system is downgrade secure over the operations defined. We now introduce a new operation that permits downgrading only by users with a current downgrader role.

SetClassif(ent,level), defined in Table 7, checks to see that downgrader is in the role set of *CurrentUser* when level does not dominate the classification of the entity before the operation. Since

Table 7 — Setting Classifications

Gate	Description
SetClassif (*ent*, *level*)	If setting the classification of *ent* to *level* would violate the classification hierarchy assertion, do nothing. Otherwise, if *level* dominates the classification of *ent* then set the classification of *ent* to *level*. Otherwise, only if *CurrentUser* has a current role of downgrader, set the classification of *ent* to *level*. (Note: each level dominates itself.)

SetClassif alters classifications, it must also include checks to ensure the invariance of Hier (see Classification Hierarchy). This can done by inspecting all of the entities that *ent* contains (and those that contain *ent*) comparing their levels to that of *ent*.

Release Secure

> *No draft message can be released except by a user with the role of releaser. The user identifier of the releaser must be recorded in the releaser field of the draft message.*

To enforce this assertion, the entity monitor must be able to distinguish between draft and released messages as well as the releaser field of the message. This is done by tagging each entity with a type. If the type is RM, that entity also has a releaser field. The entity must remain type RM and, henceforth, the contents of the entity's releaser field must not change. If an entity is not type RM before invoking an operation and is type RM immediately after, we require that the resultant releaser field contain the identifier of the user who invoked the operation. We also require that the operation performed be *Release*, that *Releaser* be in the current role set of the user, and that the type of the entity was DM immediately before the release operation.

All of these constraints can be enforced by the EM. We add a gate to the EM interface, the only gate that alters the entity type or releaser field of an entity. This gate performs the necessary checks. We constrain all the other gates to the EM not to modify the type or releaser of any entity. These constraints can be implemented in the EM by including the entity type and releaser field in the access controller. Table 8 defines these additional gates.

Table 8 — Access Operators for Release

Gate	Description
Release(ent)	If *ent* is not of type DM or *CurrentUser* does not have releaser in his current role set, then perform no action. Otherwise, set the type of *ent* to RM and set the releaser of *ent* to the *CurrentUser*.
IsDraft (*ent*)	True if and only if *ent* is of type DM.
IsReleased (*ent*)	True if and only if *ent* is of type RM.
Releaser (*ent*)	If *ent* is of type RM, then returns the releaser recorded for the entity; otherwise returns undefined.

COMPLETENESS

For the presented case analysis to be convincing, we must show that we have covered all the necessary cases. Our approach has been to enumerate the security assertions and for each assertion to determine what checks the gates must make to enforce that assertion. If we forget to consider a single case, it is possible that a security flaw could pass our analysis undetected. To show we have considered all cases, we present the results of the preceding analysis in Table 9 by summarizing the responsibilities of the gates defined with respect to the security assertions. The rows are labeled with the gates, and the columns are labeled with the security assertions. The table summarizes the results of how each security assertion interacts with the semantics of each gate. These were derived from statements of each security assertion as an invariant in terms of the gates. It is possible for each assertion to identify the gates that must perform checking to uphold the assertion and the gates that are necessary to perform those assertion checks.

One can also use the table to analyze the claims made about gates and the way they interact with assertions. For example, the first column describes the classification hierarchy assertion. We have claimed that *SetSubEnt* and *SetClassif* must perform security checks to guarantee that assertion. These checks depend on information provided by *ExistsEnt*, *SubEnt*, and *Classif.* No other gates interact with the security checking performed for the classification hierarchy.

At this point, the entries in the table are justified by an intuitive understanding of the security properties and the semantics of the functions performed by each gate. It should be possible to provide a stronger basis for the table by formalizing the semantics of each gate mathematically and basing the analysis on the formal presentation of the security model. We have taken steps in this direction in a separate formal analysis [13], and the approach appears quite promising.

DISCUSSION

We have described an architecture for implementing the SMMS security model. Over the course of producing this design, a number of issues were raised that had to be resolved before proceeding with the design.

We had to decide whether or not the operations that appear in the triples making up access sets should be generic like the EM access operations (e.g., *ValueOf(ent))*, very specific to the semantics of the user commands (e.g., *SendMessage*, *ForwardMessage*), or both. We decided that these operations should be specific to the semantics of the user commands for several reasons. Operations must make sense to the end users of the system. It is not clear that every EM call has an intuitive meaning to end users, especially after the call is embedded in a more complex client program such as *ForwardMessage*. Access sets must be understandable to the end users of the system, or it is unreasonable to assume that the users will be able to manage them effectively. If all EM calls can occur in access sets, the access set could become unwieldy or even incomprehensible.

Much of the analysis presented here was initially performed at the level of the hardware instruction set manipulating pointer registers and segment descriptors. It was later adapted to an implementation on top of an ABM. This gave us the flexibility to take advantage of a variety of protection mechanisms that might be provided on actual base machines. It should be possible to implement the ABM on bare hardware. This would require greater effort than building on top of an existing operating system but could yield greater assurance. We judge the level of effort required for implementation on bare hardware to be beyond the resources of our project. We also judge that a prototype that took advantage of an existing operating system would be of greater interest to potential builders of secure applications.

Table 9 — Gates and Security Checking

Gate	**Security Assertion**									
	Hier	Acc	Label	View	CCR	Copy	Trans	Set	Down	Rel
NewEnt	–	–	–	–	–	–	–	–	–	–
SetSubEnt	C	–	C	C	–	–	C	–	–	–
ExistsEnt	D	D	D	D	D	D	D	D	D	D
SubEnt	D	–	D	D	D	–	D	–	–	–
Classif	D	–	D	D	D	D	D	D	D	–
CreateUser	–	–	–	–	–	–	–	–	–	–
ExistsUser	–	D	–	D	D	–	D	D	D	D
SetUserClearance	–	–	–	C	–	–	–	C	–	–
Clearance	–	–	–	D	D	–	D	–	–	–
SetActiveRoles	–	–	–	–	–	–	–	C	–	–
ActivRoles	–	D	–	–	–	–	–	D	D	D
SetAuthRoles	–	–	–	–	–	–	–	C	–	–
AuthRoles	–	–	–	–	–	–	–	–	–	–
SetAccessSet	–	–	–	–	–	–	–	–	–	–
AccessSet	–	D	–	–	–	–	–	–	–	–
CurrentUser	–	D	–	D	D	–	D	D	D	D
InstallOp	–	–	–	–	–	–	–	–	–	–
OP	–	D	–	–	–	–	–	–	–	–
InstallArg	–	C	–	–	C	C	–	–	–	–
Arg	–	D	–	–	–	D	–	–	–	–
Ref	–	–	–	–	D	–	D	–	–	–
Permits	–	–	–	–	–	D	–	–	–	–
Invoke	–	–	–	–	–	–	–	–	–	–
SetValueOfArg	–	–	–	–	–	–	–	–	–	–
ValueOfArg	–	–	–	–	–	–	–	–	–	–
MakeDev	–	–	C	C	–	–	–	–	–	–
IsDevice	–	–	D	D	–	–	D	–	–	–
CurrentTerminal	–	–	–	D	–	–	–	–	–	–
SetCCR	–	–	–	–	–	–	–	–	–	–
IsCCR	–	–	–	–	D	–	D	–	–	–
CanDisplayDirect	–	–	–	–	–	–	D	–	–	–
SetClassif	C	–	C	C	–	–	–	C	C	–
Release	–	–	–	–	–	–	–	–	–	C
IsDraft	–	–	–	–	–	–	–	–	–	D
IsReleased	–	–	–	–	–	–	–	–	–	D
Releaser	–	–	–	–	–	–	–	–	–	D

Key:	C	The gate performs some special checking to ensure that the security assertion holds.
	D	System checks enforcing the security assertion depend on information this gate returns.
	–	Neither C nor D; the gate does not affect the validity of the security assertion.

The ABM description is deliberately noncommittal about the identity of the system objects it supports. We leave it up to the later design and implementation of the ABM whether or not these system objects correspond to files, segments, or other objects. We may find it convenient to use files as system objects for an ABM constructed on top of a UNIX-like operating system such as IBM's B2 Secure XENIX [14], as the SMMS project is planning. If a base machine provides an operating system oriented toward segments or consists of conventional bare hardware such as the iAPX 286 [15], we may find it more convenient to use segments instead.

CONCLUSIONS

We outline a security architecture that is being used in the design of a prototype military message system. The architecture will continue to be refined as our prototyping progresses, but the basic structure is sound and will remain reasonably stable. It should be possible for a reader to determine the mechanisms that enforce a particular security property and the arguments that need to be made about the implementation or design of the system. This is necessary to assure an evaluator that the system will behave in accordance with its stated security requirements.

The security arguments of this architecture are informally described. To gain stronger assurance, the structure described in this document must be formalized, and a correspondence to the security model must be proved.

ACKNOWLEDGMENTS

Many individuals contributed to the work reported here. Carl Landwehr suggested that we write a security architecture and explain what role it should play. Sekar Chandersekaran and Gary Luchenbaugh, of IBM, provided insightful comments. The design benefited from talks with Rob Jacob, Alan Bull, Karen Cross, Jon Weissman, Alix Evans, and others. A thorough review by Richard Hale improved the organization of this report. For funding for our work, we are grateful to H.O. Lubbes, of the Space and Naval Warfare Systems Command, and to the National Computer Security Center.

REFERENCES

1. D.E. Bell and L.J. LaPadula, "Secure Computer Systems: Unified Exposition and Multics Interpretation," MTR-2997, MITRE Corp., Bedford, MA, Mar. 1976. Available as NTIS ADA 023 588.

2. M. Cornwell and R. Jacob, "Structure of a Rapid Prototype Secure Military Message System," Proc. of the 7th DOD/NBS Computer Security Conference, Gaithersburg, MD, Sept. 1984, pp. 48-57.

3. C. Landwehr, C. Heitmeyer, and J. McLean, "A Security Model for Military Message Systems," *ACM Trans. Computer Syst.* (Aug. 1984). Also published as NRL Report 8606, May 1984.

4. D.E. Denning, "A Lattice Model of Secure Information Flow," *Commun. ACM* **18**(5), 236-243 (1976).

5. J. McLean, "Reasoning about Security Models," Proc. 1987 IEEE Symposium on Security and Privacy, Apr. 1987, IEEE Computer Society Press pp. 123-131.

6. B. Tretick, M. Cornwell, C. Landwehr, R. Jacob, and J. Tschohl, "User's Manual for the Secure Military Message System M2 Prototype," NRL Memorandum Report 5757, Mar. 1986.

7. C. Heitmeyer and M. Cornwell, ''Specifications for Three Members of the Military Message System (MMS) Family,'' NRL Memorandum Report 5645, Sept. 1985.

8. S.H. Wilson, N.C. Goodwin, E.H. Bersoff, and N.M. Thomas III, ''Military Message Experiment—Vol. I. Executive Summary,'' NRL Memorandum Report 4454, Mar. 1982.

9. J. Rushby and B. Randell, ''A Distributed Secure System,'' *Computer* **16**(7), July 1983, pp. 55-67.

10. B.W. Lampson, ''Protection,'' Proc. of the Fifth Princeton Symposium on Information Sciences and Systems, Princeton University, Mar. 1971, pp. 437-443, reprinted in *Operating Syst. Rev.* **8**(1), 18-24 (1974).

11. J.H. Saltzer and M.D. Schroeder, ''The Protection of Information in Computer Systems,'' *Proc. IEEE* **63**(9), Sept. 1975.

12. M. Cornwell, A. Evans, and J. Quinn, ''Interface Specifications for the Entity Monitor Module,'' NRL Technical Memorandum 5590-76:MC:mc, Release 1.24, Mar. 1989.

13. M. Cornwell, ''Security Architecture Proof for the SMMS Full Scale Prototype,'' working document.

14. C.S. Chandersekaran, R.S. Chapman, L.J. Dotterer, M.S. Hecht, W.D. Jiang, G.L. Luchenbaugh, and N. Vasudevan, ''On the Design and Implementation of Secure Xenix Workstations,'' Proc. of the 1986 IEEE Symposium on Security and Privacy, Apr. 1986, pp. 102-117.

15. Intel Corporation, *iAPX 286 Preliminary User's Manual*, June 1981.

Appendix

GLOSSARY

abstract base machine (ABM)

The lower level construct upon which the domain structure is implemented. The ABM provides the primitive security functions necessary to implement higher level structures for the Secure Military Message System (SMMS) security model.

access set

A set of triples of the form (u, op, k), where u is a user identifier or role, op is an operation, and k is a parameter position to op. Each entity is associated with an access set. To perform an operation op with entity e in the kth parameter position, a triple of the form (u, op, k) must appear in the access set of e, where u is the user identifier requesting the operation or one of the user's current roles.

authorized roles

The set of roles in which a user is authorized to act.

classification

A security level denoting the sensitivity of information.

classification hierarchy

A security invariant stating that the classification of any entity must dominate the classification of any entities it contains.

clearance

The security level granted to the user based on background and security checks.

contain

A relation between entities. An entity may contain other entities, which in turn may contain still more entities.

containment hierarchy

The containment relation between entities. (See contain).

current roles

The set of roles in which a user is currently acting.

direct reference

A reference to an entity consisting of the unique identifier for that entity only.

domain

A named context in which a program can execute that determines the accesses allowed by that program. The essential difference between domains is that certain accesses are allowed in one domain that are not allowed in another. Whether a particular program can perform a particular access depends on the domain in which it is executing.

dominates

(As a relation between security levels) Given two security levels A and B, A dominates B if and only if the sensitivity level of A is at least as high as the sensitivity level of B, and the compartment names of B are a (possibly improper) subset of the compartment names of A.

(As a relation between entities) Given two entities *el* and *e*2, *e*1 dominates *e*2 if and only if the classification of *e*1 dominates the classification of *e*2.

This relationship forms a mathematical lattice as described by Denning [3].

entities

Objects in the system that hold the information we want to protect. Each entity is associated with a classification denoting the sensitivity of the information it holds. Entities may contain other entities. Entities are also associated with access sets.

entity monitor domain

A particular domain within the domain structure of the SMMS that carries most of the security responsibilities for the system. The entity monitor domain encapsulates the majority of the system state.

gates

(From A to B) The system objects in domain B that are executable by processes in domain A. These gates allow a process executing in domain A to have some effect on system objects in domain B.

history

A record of the sequence of the states through which the system may pass as users issue requests [2].

indirect reference

A reference to an entity that refers to the entity in terms of some entity that contains it. For example, "the third entity contained in entity MF134."

invariant properties

Those properties that are true before and after each change to the state of the system.

protected subsystem

A subsystem built as part of a larger system that once constructed has the ability to protect itself (its code and data structures) from tampering by the rest of the system [11].

releaser field

An attribute of any entity of type released message (RM). This attribute holds the user identifier of the user who released the message.

request

A $N+1$−tuple $\langle op, xl, \ldots, xN \rangle$ consisting of an operation *op* and a list of parameters *xl*, . . . ,*xN*. The parameters may be references to entities, user identifiers, or arbitrary strings. Requests are intended to correspond to the user commands of the application software.

roles

A special authority by which a user may act. Roles are used in addition to other information to determine what accesses are permitted.

secure state

A system state exhibiting certain properties determined by inspecting the state in isolation. (See Definition [2].)

security architecture

That part of the system structure responsible for providing security. It should cover enough of the system structure to argue convincingly that the system is secure.

security assertions

Part of the security requirements of a system. Those properties that the system can and must enforce to be considered secure.

security invariants

Security assertions that are stated as invariant properties of the system state.

security level

Consists of a sensitivity level and a set of compartment names.

sensitivity level

One of the four DoD sensitivity levels: TOP SECRET, SECRET, CONFIDENTIAL, UNCLASSFIED.

state

(Of the system in the SMMS security model) In the formal model, the system state consists of a triple (u,e,lo) where u is a function mapping users-to-user identifiers, e is a mapping from reference to entities, and lo is a function mapping users to output devices [2].

system objects

Provided by the ABM (e.g. files, segments, ports, message queues) but may differ from one base machine to another.

system security officer (SSO)

A role that is required for certain security-related operations. A user must have an SSO among its current roles to modify clearances, and authorize role sets, and perform other such security-related operations.

system structure

A set of design decisions constraining the potential systems under consideration. Intuitively, one starts the design process with no design decisions, so the set of potential systems encompasses all systems. As design decisions are made, systems not in conformance with those design decisions are eliminated from consideration. These design decisions impose a structure on all the systems remaining under consideration. Reasoning from a given set of design decisions, we arrive at conclusions that hold for all the systems under consideration.

system transform

A function that takes a user identifier, a request, and a state and maps them to a new state. We normally write this as T $(u,\ i,\ s) = s*$. The system transform captures the intuitive notion of what behavior is allowed by the system.

trusted processes

Processes that are allowed to violate the security constraints placed on other processes. A term from Bell-LaPadula [1] based security models.

unique identifier

A string associated with an entity, that in a given state, uniquely identifies the entity.

user

The human user of a system. A user is associated with a clearance. The system changes state based on requests by users:

user identifier

A string used to denote a user. Each user is associated with a unique user identifier.

value

A string (of characters or bits) associated with an entity. This value has no specific structure so far as the security model is concerned. This value is independent of the other attributes of an entity such as classification and the entities it contains.